Chapter 3 Agile

3.1 Agile methods

3.2 Agile Development techniques

3.3 Agile Project Management

3.4 Scaling Agile Methods

3.1 Agile methods

Historical Context

- In the 1980s and 1990s, a plan-driven approach was popular, especially for large, complex systems.
- Such heavyweight approaches were found to be inefficient for small and medium-sized projects due to high overheads.

Emergence of Agile Methods

- Developed in the late 1990s as a response to dissatisfaction with heavyweight, plan-driven approaches.
- Focus on the software itself, minimizing design and documentation overheads.
- Suited for rapidly changing requirements and aims to deliver working software quickly.

Agile Manifesto Principles

1. **Customer Involvement**: Customers should be closely involved to provide and prioritize new system requirements.
2. **Embrace Change**: The system should be designed to accommodate changing requirements.
3. **Incremental Delivery**: Software should be developed and delivered in increments, based on customer-specified requirements.
4. **Maintain Simplicity**: Focus on simplicity in both the software and the development process.
5. **People, not Process**: Exploit the skills of the development team; avoid prescriptive processes.

Core Values (from Agile Manifesto)

- **Individuals and Interactions** over processes and tools

- **Working Software** over comprehensive documentation
- **Customer Collaboration** over contract negotiation
- **Responding to Change** over following a plan

Ideal Use-Cases for Agile Methods

1. **Product Development**: Particularly effective for small or medium-sized products for sale.
2. **Custom System Development**: Effective when the customer is committed to being involved and when there are few external stakeholders.

Communication and Team Structure

- Agile methods work well when there is continuous communication between stakeholders and the development team.
- Best suited for co-located teams where informal communication is effective.

Limitations

- Less suited for systems that are tightly integrated with other systems being developed simultaneously.
- Not ideal for situations requiring coordination of parallel development streams.

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Embrace change	Expect the system requirements to change, and so design the system to accommodate these changes.
Incremental delivery	The software is developed in increments, with the customer specifying the requirements to be included in each increment.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.
People, not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.

Figure 3.2 The principles of agile methods

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

3.2 Agile Development techniques

Historical Background

- Extreme Programming (XP), coined by Kent Beck, significantly influenced the agile development culture.
- XP aimed to push recognized good practices like iterative development to "extreme" levels.

Core Concepts in XP

1. **Requirements as User Stories**: Requirements are expressed as scenarios or user stories implemented as tasks.
2. **Frequent Releases**: Short time gaps between system releases, focusing on incremental development.
3. **Test-First Approach**: Tests are developed for each task before writing the actual code.
4. **Pair Programming**: Programmers work in pairs to cross-verify and support each other.

XP Practices Aligned with Agile Manifesto Principles

1. **Incremental Development**: Small, frequent releases based on simple customer stories.
2. **Customer Involvement**: Constant customer engagement, with customer representatives defining acceptance tests.
3. **People Over Process**: Practices like pair programming, collective ownership, and sustainable work hours.
4. **Embracing Change**: Regular releases, test-first development, and continuous integration.
5. **Maintaining Simplicity**: Constant refactoring and using simple designs that are easily adaptable.

Extreme Programming Practices

1. **Collective Ownership**: All developers are responsible for all code; anyone can change anything.
2. **Continuous Integration**: Completed tasks are immediately integrated into the whole system, passing all unit tests.
3. **Incremental Planning**: Requirements are recorded on "story cards," prioritized based on time and importance.
4. **On-Site Customer**: A customer representative is part of the development team, responsible for conveying system requirements.
5. **Pair Programming**: Developers work in pairs, mutually verifying and supporting each other.
6. **Refactoring**: Code is continuously improved for simplicity and maintainability.
7. **Simple Design**: Only enough design is carried out to meet the current requirements.
8. **Small Releases**: Frequent and incremental releases that add functionality to the initial release.
9. **Sustainable Pace**: Avoiding excessive overtime to maintain code quality and productivity.
10. **Test-First Development**: An automated unit test framework is used to write tests before implementing functionality.

Adaptability and Challenges

- XP as originally proposed is difficult to integrate into the management practices and culture of most businesses.
- Companies often pick and choose XP practices that best suit them, sometimes integrating them into other agile methods like Scrum.
- XP's most significant contribution is likely the set of agile development practices it introduced.

Principle or practice	Description
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Incremental planning	Requirements are recorded on "story cards," and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development "tasks." See Figures 3.5 and 3.6.
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Refactoring	All developers are expected to refactor the code continuously as soon as potential code improvements are found. This keeps the code simple and maintainable.
Simple design	Enough design is carried out to meet the current requirements and no more.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Sustainable pace	Large amounts of overtime are not considered acceptable, as the net effect is often to reduce code quality and medium-term productivity.
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.

Figure 3.4 Extreme programming practices

4. Change is embraced through regular system releases to customers, test-first development, refactoring to avoid code degeneration, and continuous integra

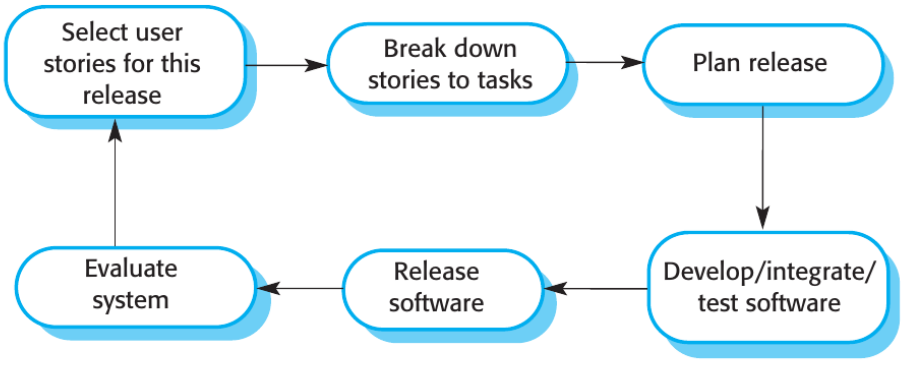

Figure 3.3 The XP release cycle

3.2.1 User Stories

Overview and Purpose

- User stories are scenarios of use developed to encapsulate customer needs.
- They integrate requirements elicitation with agile development, replacing separate requirements engineering activities.

Creation and Usage

1. **Collaborative Development**: The system customer and development team collaborate to develop a "story card" that briefly describes a user story.
2. **Task Breakdown**: Each story card is broken down into tasks, with estimations for effort and resources required.
3. **Customer Prioritization**: The customer prioritizes the stories based on immediate business value.
4. **Short Iterations**: The aim is to implement functionality from prioritized stories in about two weeks, aligning with the next system release.

Advantages

- **Dynamic Adaptability**: Unimplemented stories can change or be discarded as requirements evolve.
- **User Engagement**: User stories are more relatable than conventional requirements documents, fostering better user involvement.

Challenges and Limitations

1. **Completeness**: It's difficult to assess if enough user stories have been developed to cover all essential requirements.
2. **Accuracy**: User stories may not capture all aspects of an activity, especially if experienced users omit details.

Application in Requirements Elicitation

- User stories can serve as an initial step in pre-development requirements elicitation, a topic discussed further in future chapters.

Prescribing medication

Kate is a doctor who wishes to prescribe medication for a patient attending a clinic. The patient record is already displayed on her computer so she clicks on the medication field and can select 'current medication', 'new medication' or 'formulary'.

If she selects 'current medication', the system asks her to check the dose; If she wants to change the dose, she enters the new dose then confirms the prescription.

If she chooses 'new medication', the system assumes that she knows which medication to prescribe. She types the first few letters of the drug name. The system displays a list of possible drugs starting with these letters. She chooses the required medication and the system responds by asking her to check that the medication selected is correct. She enters the dose then confirms the prescription.

If she chooses 'formulary', the system displays a search box for the approved formulary. She can then search for the drug required. She selects a drug and is asked to check that the medication is correct. She enters the dose then confirms the prescription.

The system always checks that the dose is within the approved range. If it isn't, Kate is asked to change the dose.

After Kate has confirmed the prescription, it will be displayed for checking. She either clicks 'OK' or 'Change'. If she clicks 'OK', the prescription is recorded on the audit database. If she clicks on 'Change', she reenters the 'Prescribing medication' process.

Figure 3.5 A "prescribing medication" story

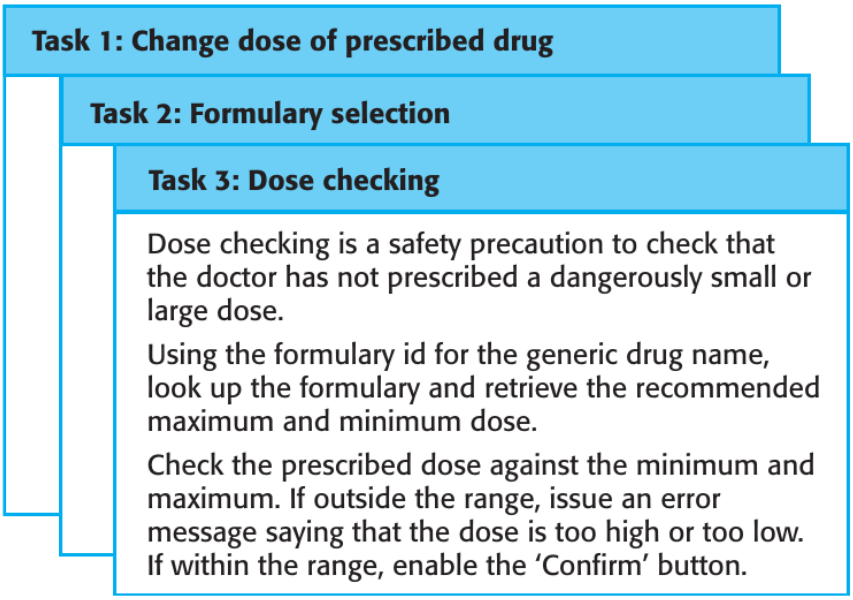

Figure 3.6 Examples of task cards for prescribing medication

3.2.2 Refactoring

Concept and Rationale

- Refactoring is the practice of continually improving software code for better structure and readability.
- It deviates from the traditional software engineering principle of "designing for change," arguing that such anticipation often results in wasted effort.

Process

1. **Immediate Implementation**: When developers see code that can be improved, they make these improvements immediately, regardless of immediate necessity.
2. **Avoiding Structural Decay**: Refactoring counters the software structure degradation that naturally occurs in incremental development.

Types of Refactoring

- Examples include reorganizing class hierarchies, renaming attributes and methods, and replacing duplicated code with library methods.

Tools and Support

- Development environments often include tools that simplify the process of finding code dependencies and making global modifications.

Benefits

- Keeps the software easy to understand and modify, which is particularly useful as new requirements emerge.

Challenges and Limitations

1. **Development Pressure**: Sometimes, new feature implementation takes precedence over refactoring, causing a delay in improvements.
2. **Architectural Limitations**: Some new features and changes may require architectural modifications that cannot be addressed solely through code-level refactoring.

3.2.3 Test-first Development (TDD)

Overview

- Originated from Extreme Programming (XP), test-first development is an approach where tests are written before the code that needs to be tested.
- It addresses challenges related to testing in incremental development methodologies.

Key Features in XP

1. **Test-First Development**: Tests are written before the code.
2. **Incremental Test Development from Scenarios**: Tests are developed based on user stories or scenarios.
3. **User Involvement**: Users are involved in test development and validation.
4. **Automated Testing Frameworks**: Use of frameworks like JUnit for automated testing.

Advantages

- Helps in discovering problems during development.
- Defines an interface and a specification of behavior implicitly.
- Helps in understanding specifications thoroughly before implementation.
- Avoids "test-lag" where implementation outpaces testing.

Test Development Process

- User stories are broken down into tasks, each generating one or more unit tests.
- Acceptance tests are developed with customer involvement to ensure the system meets real needs.

Role of Automation

- Test automation is essential for quickly and easily executing a large set of tests.
- Catches issues immediately when new functionality is added.

Challenges and Limitations

1. **Incomplete Tests**: Programmers may skip writing comprehensive tests.
2. **Difficulty in Incremental Testing**: Some elements, like complex user interfaces, are hard to test incrementally.
3. **Test Coverage**: Despite a large set of tests, it is difficult to ensure complete coverage, leaving potential bugs undetected.

Generalization

- The philosophy of test-first has evolved into more general test-driven development techniques.

Test 4: Dose checking

Input:
1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:
1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:
OK or error message indicating that the dose is outside the safe range.

Figure 3.7 Test case description for dose checking

3.2.4 Pair Programming

Overview

- Pair programming is an Extreme Programming (XP) practice where two developers work together at the same computer.
- Pairs are created dynamically, allowing all team members to collaborate throughout the development process.

Advantages

1. **Collective Ownership and Responsibility**: Promotes the idea that the entire team owns the software and shares responsibility for its issues.
2. **Informal Code Review**: Acts as a less formal but quicker alternative to traditional code inspections and reviews, aiding in error discovery.
3. **Encourages Refactoring**: The practice supports immediate improvements to the software structure, which benefits the team as a whole.

Efficiency Considerations

- Contrary to the belief that pair programming would halve productivity, some studies suggest comparable productivity levels to individuals working alone. This is attributed to fewer false starts and less rework.

Adoption in Industry

- Companies have mixed views on pair programming; some avoid it, while others use a mix of pair and individual programming, often pairing experienced programmers with less experienced ones.

Empirical Studies

- Studies on the effectiveness of pair programming have had mixed results:
  - Some research found comparable productivity and fewer errors (Williams et al. 2000).
  - Other studies found a loss of productivity but some quality benefits (Arisholm et al. 2007).

Risk Mitigation

- The knowledge sharing that occurs during pair programming can reduce project risks, especially when team members leave, potentially making the practice worthwhile despite its overhead.

3.3 Agile Project Management

Overview

- Agile project management diverges from traditional plan-driven approaches, aiming for more flexibility and responsiveness.
- Scrum has emerged as the most popular framework for organizing agile projects, providing a degree of external visibility.

Scrum Framework

- Focuses on project organization, not specific development practices like pair programming or test-first development.
- Uses unique terminology like "ScrumMaster" instead of "Project Manager" to distinguish itself from traditional methods.

Scrum Process Components

1. **Product Backlog**: A prioritized list of features, requirements, and other tasks for the development team.
2. **Sprint**: A time-boxed development cycle, usually 2-4 weeks long.

3. **Sprint Backlog**: Specific tasks chosen for the current sprint based on priority and previous sprint velocity.
4. **Daily Meetings (Scrums)**: Short meetings to review progress and re-prioritize work.
5. **Sprint Review**: At the end of each sprint, team reviews work and process for improvement.

Sprint Planning

- Product Owner prioritizes backlog items.
- Team estimates time required for the highest-priority items based on previous sprints' velocities.
- Unfinished items return to the product backlog.

Team Coordination

- Uses Scrum board for daily interactions.
- All team members have visibility into the work being done.

Review and Feedback Loop

- End-of-sprint review serves dual purposes: process improvement and product state assessment.

ScrumMaster Role

- Although not formally a project manager, ScrumMasters often assume this role in organizations with a conventional management structure.

Benefits of Using Scrum

1. Manageable and understandable product chunks.
2. Handles unstable requirements effectively.
3. Enhances team communication and morale.
4. Offers customers on-time delivery of increments.
5. Builds trust between customers and developers.

Distributed Scrum

- Scrum is adaptable for distributed teams and multi-team environments, accommodating global development needs.

Challenges

- Originally designed for co-located teams, adaptations are needed for distributed teams.

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than seven people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be "potentially shippable," which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of "to do" items that the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories, or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development, and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2 to 4 weeks long.
Velocity	An estimate of how much product backlog effort a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

Figure 3.8 Scrum terminology

the team. To address this issue, the Scrum agile method was developed (Schwaber

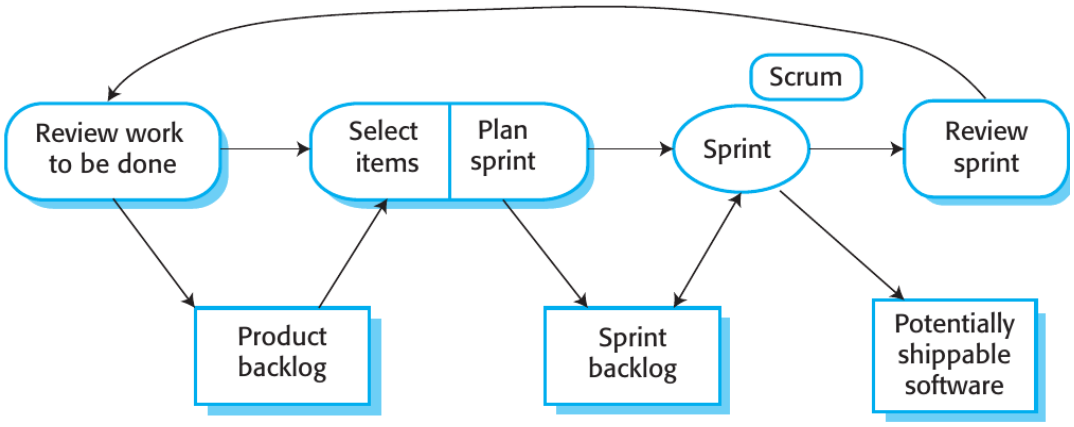

Figure 3.9 The Scrum sprint cycle

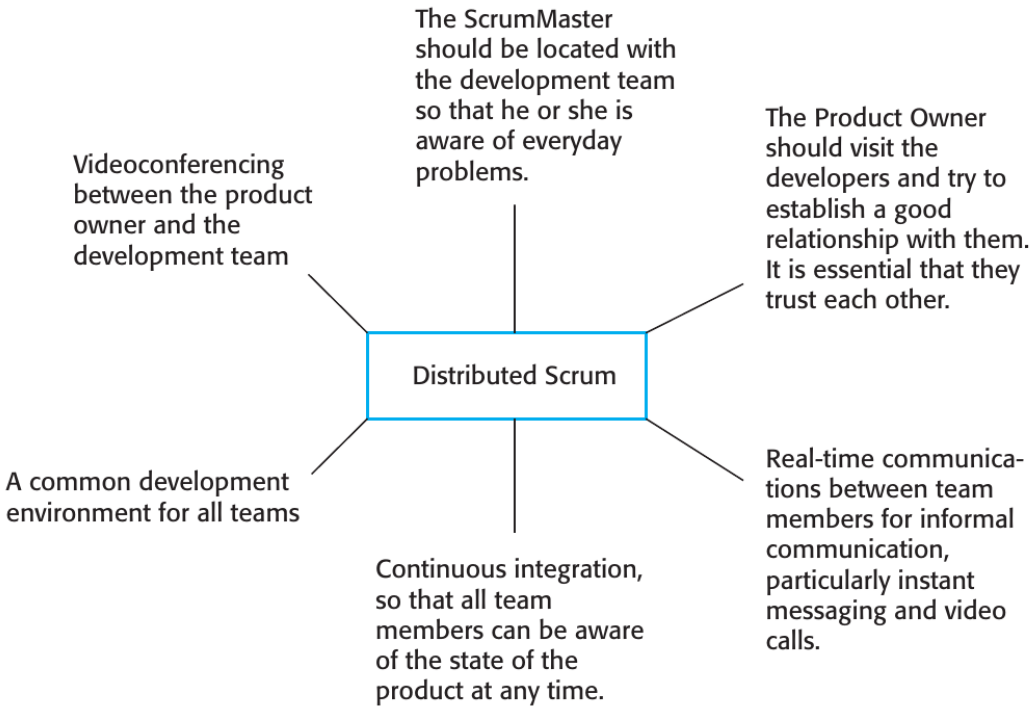

Figure 3.10 Distributed Scrum

3.4 Scaling Agile Methods

Initial Adoption

- Agile methods were initially designed for small teams working on small to medium-sized projects in the same physical location.
- Small companies with less bureaucracy were the early adopters.

Need for Scaling

- The demand for faster, customer-focused software delivery is equally relevant to larger systems and companies.

Facets of Scaling

1. **Scaling Up**: Adapting agile methods to manage large software projects that cannot be handled by a single small team.
2. **Scaling Out**: Extending the use of agile methods across various departments in a large organization with extensive software development experience.

Challenges in Scaling

- Larger organizations often have to deal with scaling up and scaling out simultaneously, especially when they win contracts for large projects.

Productivity and Defects

- There are claims of substantial productivity gains and defect reductions through the use of agile methods.

- Scott Ambler suggests that for large systems and organizations, the productivity improvement is likely to be around 15% over 3 years, along with similar reductions in defects.

Summary

- Scaling agile methods is a complex but necessary evolution to fit the needs of larger projects and organizations. Expectations of productivity gains should be realistic.

3.4.1 Practical problems with agile methods

General Suitability

- Agile methods are highly effective for software products and apps.
- They may not be appropriate for embedded systems, large and complex systems, or ongoing software maintenance.

Challenges for Large, External Projects

1. **Contractual Issues**: The informality of agile development clashes with the legal contracts often used in large projects.
2. **Maintenance Costs**: Agile is geared more toward new development, whereas most costs in large companies stem from maintenance.
3. **Global Teams**: Agile is designed for small, co-located teams, but many modern projects involve globally distributed teams.

Specific Issues

- **Lack of Definitive Requirements**: Agile's evolving requirements don't fit well into traditional contracts that define specific deliverables.

Maintenance Issues

1. **Lack of Documentation**: Agile methods often lack formal documentation, making it challenging to assess the impact of proposed system changes.
2. **Customer Involvement**: It's difficult to keep customers engaged during the maintenance phase.
3. **Team Continuity**: Agile methods rely on the team's collective memory and expertise, which can be lost if the team is disbanded or changed.

Summary

- While agile methods excel in specific scenarios, they have limitations and challenges when applied to large projects, maintenance, or distributed teams. These include contractual complexities, the focus on new development over maintenance, and issues related to documentation and team continuity.
-

3.4.2 Agile and plan-driven methods

Introduction

- Integration of agile and plan-driven methods is essential for scaling agile in larger companies.
- Early agile enthusiasts resisted plan-driven approaches, but adaptations are necessary for various organizational and technical contexts.

Deciding Between Agile and Plan-Driven

- A balanced approach often combines elements of both plan-driven and agile methods.
- Factors influencing the balance include system attributes, development team characteristics, and organizational context.

System-Related Issues

1. **System Size**: Agile is most effective for small, co-located teams. Larger systems may require a plan-driven approach.
2. **System Type**: Systems requiring extensive pre-implementation analysis (e.g., real-time systems) may be better suited for a plan-driven approach.
3. **System Lifetime**: Long-lasting systems may require more documentation for long-term maintenance.
4. **External Regulation**: Systems subject to regulatory approval may require detailed documentation.

Team-Related Issues

1. **Skill Levels**: Agile may require higher skill levels. In less skilled teams, a separation of design and implementation roles may be needed.
2. **Team Organization**: Distributed or outsourced teams may require more design documentation.
3. **Technological Support**: The availability of adequate development tools can influence the need for documentation.

Organizational Issues

1. **Contractual Requirements**: A detailed specification may be necessary for contractual reasons, favoring a plan-driven approach for requirements engineering.
2. **Incremental Delivery**: Feasibility depends on customer availability and willingness to participate.
3. **Cultural Factors**: Traditional engineering organizations may be more comfortable with plan-driven methods due to their extensive design documentation.

Conclusion

- The label of "agile" or "plan-driven" is less important than choosing the methods that are most effective for the specific type of system being developed.

Principle	Practice
Customer involvement	This depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Often, customer representatives have other demands on their time and cannot play a full part in the software development. Where there are external stakeholders, such as regulators, it is difficult to represent their views to the agile team.
Embrace change	Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.
Incremental delivery	Rapid iterations and short-term planning for development does not always fit in with the longer-term planning cycles of business planning and marketing. Marketing managers may need to know product features several months in advance to prepare an effective marketing campaign.
Maintain simplicity	Under pressure from delivery schedules, team members may not have time to carry out desirable system simplifications.
People, not process	Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods and therefore may not interact well with other team members.

Figure 3.11 Agile principles and organizational practice

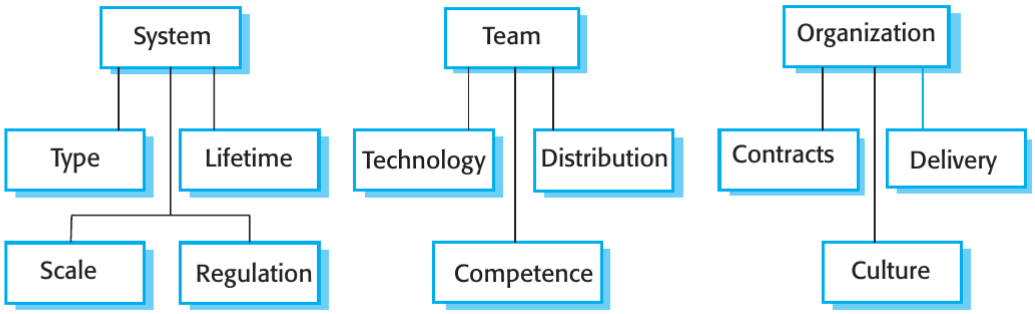

Figure 3.12 Factors influencing the choice of plan-based or agile development

3.4.3 Agile methods for large systems

Complexity Factors in Large Systems

1. **Systems of Systems**: Different teams in different locations work on sub-systems, lacking a holistic view.
2. **Brownfield Systems**: Large systems often interact with existing systems, requiring complex negotiations and less flexibility.
3. **Configuration Over Coding**: A significant portion of development is related to system configuration.
4. **External Constraints**: Rules and regulations often dictate the development processes and documentation.
5. **Long Development Time**: Maintaining a coherent team over an extended period is challenging.
6. **Diverse Stakeholders**: Involving all types of stakeholders in the development process is difficult.

Scaling Agile Frameworks

- **Scaled Agile Framework (SAFe)**: Developed to support large-scale, multi-team software development.
- **Agile Scaling Model (ASM)**: Involves stages like Disciplined Agile Delivery and Agility at Scale to adapt to organizational settings and large project complexities.

Common Features in Scaling Agile

1. **Initial Requirements**: Some early work on initial software requirements is essential but should be general.
2. **Multiple Product Owners**: Different parts of the system require different representatives who must communicate continuously.
3. **More Design and Documentation**: Up-front design and critical system documentation are necessary.
4. **Cross-Team Communication**: Various communication channels and regular meetings are vital for syncing progress across teams.
5. **Configuration Management**: Tools that support multi-team software development are essential.

Multi-Team Scrum Characteristics

1. **Role Replication**: Each team has its own Product Owner and ScrumMaster, with chief roles overseeing the entire project.
2. **Product Architects**: Each team selects an architect who collaborates on the overall system architecture.
3. **Release Alignment**: Product release dates are synchronized across teams.
4. **Scrum of Scrums**: A higher-level Scrum meeting occurs daily to discuss progress, identify problems, and plan the day's work.

Final Thoughts

- No single model suits all large-scale agile products; customization is often required.
- Agile methods need to evolve and integrate plan-based practices to tackle the complexities of large systems.

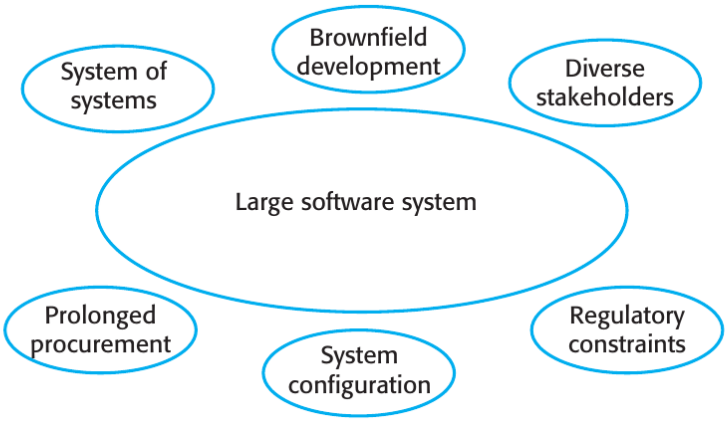

Figure 3.13 Large project characteristics

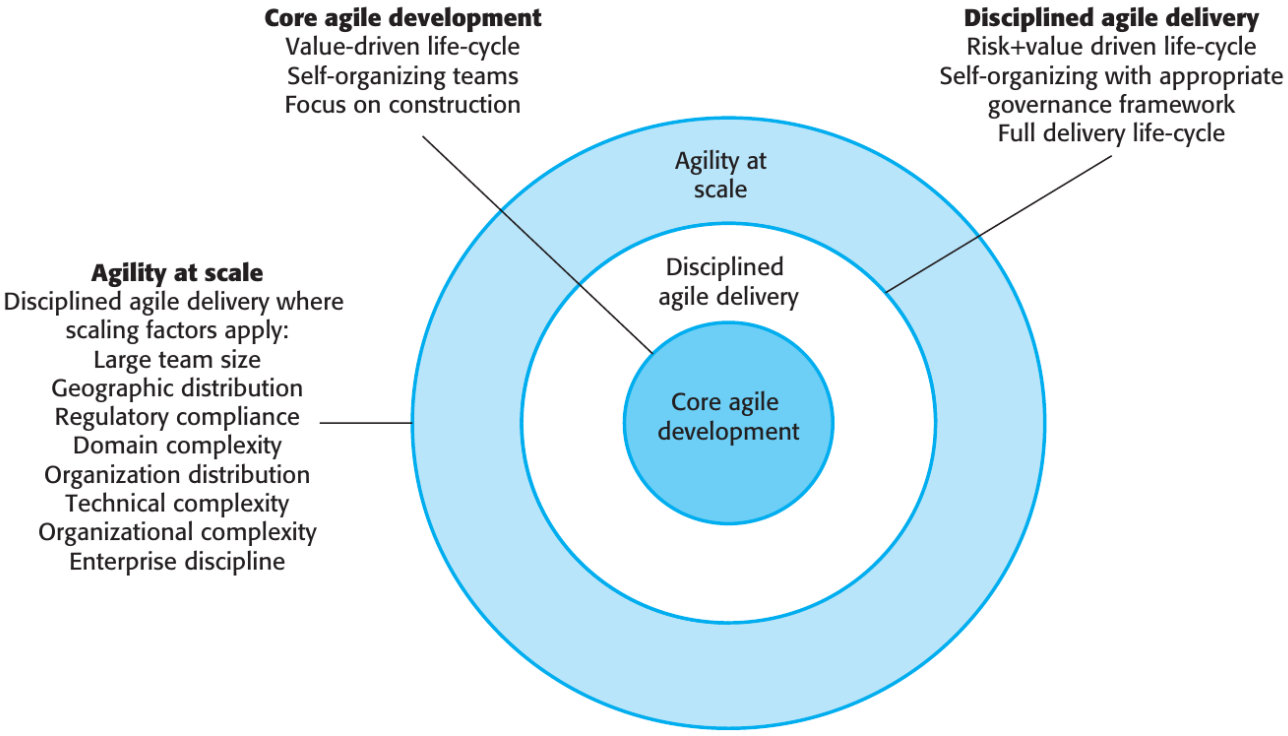

Figure 3.14 IBM's Agility at Scale model (© IBM 2010)

3.4.4 Agile methods across organizations

Agile Methods in Small vs. Large Companies

- Small software companies easily adopt agile methods due to lack of organizational bureaucracy.
- Larger companies find it challenging to scale out agile methods across the organization.

Challenges in Introducing Agile to Large Companies

1. **Reluctance from Project Managers**: Lack of experience in agile makes them hesitant to take the risk.
2. **Organizational Procedures**: Existing quality procedures and tools may conflict with agile principles.
3. **Team Skill Levels**: Agile is most effective with high-skill team members, which may not always be the case in large organizations.
4. **Cultural Resistance**: Organizations with a history of conventional systems engineering may resist agile methods.

Incompatibility with Company Procedures

- **Change Management**: Agile's flexibility in refactoring conflicts with the controlled nature of change management.
- **Testing Procedures**: External testing teams may conflict with agile's test-first approaches.

Implementing Agile in Large Organizations

- Requires a cultural change, often necessitating a change in management.

- Best introduced gradually, starting with an enthusiastic group of developers.
- Successful agile projects can serve as a starting point for broader organizational adoption.

Summary

- Agile methods are iterative development methods that focus on reducing process overheads and documentation and on incremental software delivery. They involve customer representatives directly in the development process.
- The decision on whether to use an agile or a plan-driven approach to development should depend on the type of software being developed, the capabilities of the development team, and the culture of the company developing the system. In practice, a mix of agile and plan-based techniques may be used.
- Agile development practices include requirements expressed as user stories, pair programming, refactoring, continuous integration, and test-first development.
- Scrum is an agile method that provides a framework for organizing agile projects. It is centered around a set of sprints, which are fixed time periods when a system increment is developed. Planning is based on prioritizing a backlog of work and selecting the highest priority tasks for a sprint.
- To scale agile methods, some plan-based practices have to be integrated with agile practice.
- These include up-front requirements, multiple customer representatives, more documentation, common tooling across project teams, and the alignment of releases across teams.